Android

programmation avancée

DOCDOKU
digital enterprise applications

Plan de la formation

- Construire une interface responsive
- Analyse et optimisation d'UI
- Architecture et data-binding
- Thread et Programmation réactive
- L'injection de dépendances
- Les Tests Unitaires

Construire une interface responsive

- Structurer l'interface avec ConstraintLayout
- Manipuler les assets graphiques
- Améliorer l’expérience utilisateur avec des animations simples

Construire une interface responsive

Structurer l'interface avec ConstraintLayout

- ConstraintLayout est un container de vues avec une hiérarchie “à plat”, les vues ne sont pas emboîtées mais reliées par des contraintes

- une contrainte est la connexion d’une vue à une autre vue, son parent ou même un support invisible

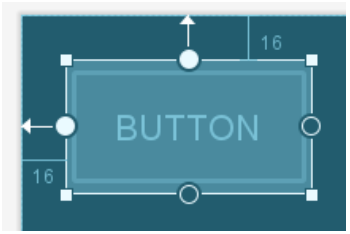
- la création des ces layouts est très flexible et adaptée à l’utilisation de l’éditeur graphique

Construire une interface responsive

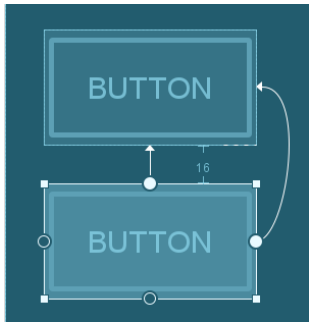
Structurer l'interface avec ConstraintLayout

Contraintes

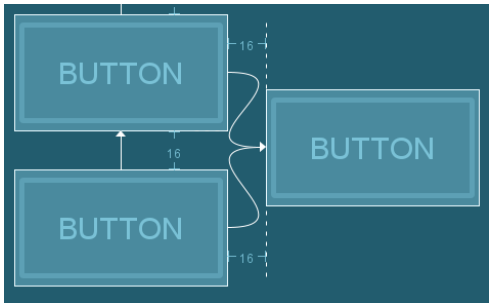
Chaque vue doit avoir 2 contraintes : une verticale et une horizontale



Une contrainte ne peut être créée qu'entre une poignée et un point d'ancrage qui partagent le même plan



Chaque poignée ne peut être utilisée que pour une seule contrainte, mais différentes contraintes peuvent rejoindre le même point d'ancrage



Construire une interface responsive

Structurer l'interface avec ConstraintLayout

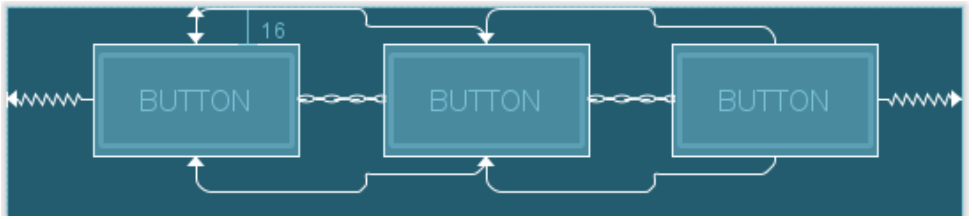
Chaîne

Une chaîne est un groupe de vues liées les unes aux autres avec des contraintes de position bidirectionnelles.

Une vue peut faire partie d'une chaîne horizontale et verticale, ce qui facilite la création de dispositions de grille flexibles.

Une chaîne ne fonctionne correctement que si chaque extrémité de la chaîne est contrainte à un autre objet sur le même axe

L'utilisation d'une chaîne n'aligne pas les vues dans sa direction.



Construire une interface responsive

Structurer l'interface avec ConstraintLayout

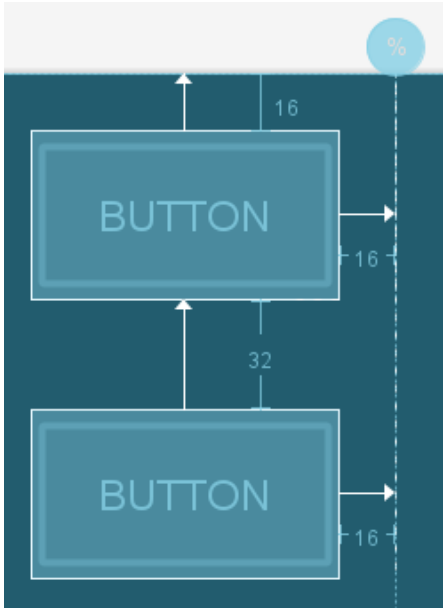
Guideline

C’est un guide visuel pour créer des contraintes communes

Aucune contrainte ne part d’une guideline

Elle n’apparaît que dans l’éditeur graphique, pas sur les écrans des appareils (sa visibilité est View.GONE)

Elle peut être placée horizontalement ou verticalement, à une distance fixe (en dp) ou selon une fraction de la dimension du parent



Construire une interface responsive

Structurer l'interface avec ConstraintLayout

Barrière

Tout comme la guideline, la barrière est invisible sur les écrans

Sa taille est définie par celle des vues qu'elle contient

Très utile dans les cas où :

- une vue doit être contrainte à plusieurs vues

- une vue est contrainte à une vue dont la visibilité peut passer de Visible à Gone (et inversement)

Construire une interface responsive

Structurer l'interface avec ConstraintLayout

Flow

Flow virtual layout permet le positionnement des widgets référencés horizontalement ou verticalement, à la manière d'une chaîne, mais avec la possibilité de passer à la ligne suivante si l'espace n'est pas suffisant.

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <androidx.constraintlayout.helper.widget.Flow
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:constraint_referenced_ids="item_1,item_2,item_3" />
    <View
        android:id="@+id/item_1"
        android:layout_width="50dp"
        android:layout_height="50dp" />
    <View
        android:id="@+id/item_2"
        android:layout_width="50dp"
        android:layout_height="50dp" />
    <View
        android:id="@+id/item_3"
        android:layout_width="50dp"
        android:layout_height="50dp" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Construire une interface responsive

Manipuler les assets graphiques

Il existe 3 principaux types de drawable :

- VectorDrawable
- BitmapDrawable
- NinePatchDrawable

Les drawables se placent dans différents dossiers

res/mipmap pour les icônes de lancement

res/drawable pour tous les autres

Construire une interface responsive

Manipuler les assets graphiques

VectorDrawable

Un VectorDrawable est un graphique vectoriel défini en XML par un ensemble de formes avec une couleur associée

Avantages

- la taille : fichier très léger comparé à une image
- la scalabilité : le même fichier est redimensionné pour différentes densités d'écran sans perte de qualité d'image

Inconvénients

- il a besoin d'être redessinés en Bitmap avant leur affichage.
- il n'est pas adapté pour l'affichage de photos
- son chargement initial peut consommer plus de cycles de CPU qu'une image normale.

Construire une interface responsive

Manipuler les assets graphiques

VectorDrawable

```
<!-- res/drawable/battery_charging.xml -->
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    android:width="24dp"
    android:height="24dp"
    android:viewportWidth="24.0"
    android:viewportHeight="24.0">
    <group
        android:name="rotationGroup"
        android:pivotX="10.0"
        android:pivotY="10.0"
        android:rotation="15.0">
        <path
            android:name="vect"
            android:fillAlpha=".3"
            android:fillColor="#FF000000"
            android:pathData="M15.67,4H14V2h-4v2H8.33C7.6,4 7,4.6 7,5.33V9h4.93L13,7v2h4V5.33C17,4.6 16.4,
                              4 15.67,4z" />
        <path
            android:name="draw"
            android:fillColor="#FF000000"
            android:pathData="M13,12.5h2L11,20v-5.5H9L11.93,9H7v11.67C7,21.4 7.6,22 8.33,22h7.33c0.74,0 1.34,
                              -0.6 1.34,-1.33V9h-4v3.5z" />
    </group>
</vector>
```

Construire une interface responsive

Manipuler les assets graphiques

VectorDrawable

En cas d'utilisation de Support library

Ajouter au build.gradle :

```
android {
    defaultConfig {
        vectorDrawables.useSupportLibrary = true
    }
}
```

et dans les layouts

```
<ImageView
    android:id="@+id/imageView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:srcCompat="@drawable/my_vector" />
```

Construire une interface responsive

Manipuler les assets graphiques

BitmapDrawable

Permet d'afficher des images de haute qualité

Supporte différents format : PNG, JPG, WebP

Mais l'affichage d'image trop grosses nuisent à la qualité de l'application. Plusieurs solutions :

- *Comme décrit dans la doc : https://developer.android.com/topic/performance/graphics/load-bitmap Le but est de mesurer les dimensions de l'image avant son chargement et la redimensionner si nécessaire.*

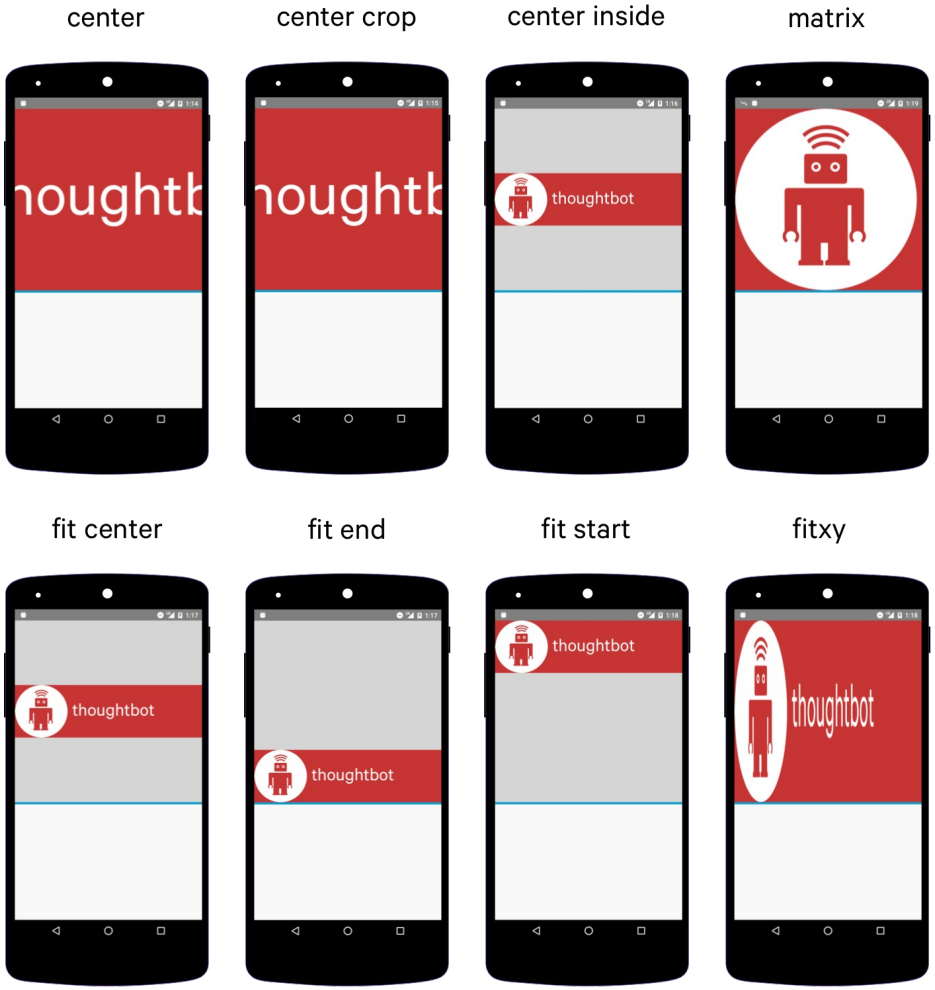
- *L'utilisation de bibliothèques externes : Glide, Picasso, mais implique une connexion internet.*

- *Compresser au max les images avant de les charger dans l'appli (en les transformant ent webP par exemple)*

Construire une interface responsive

Manipuler les assets graphiques

ScaleType



Construire une interface responsive

Manipuler les assets graphiques

NinePatchDrawable

C’est un bitmap redimensionnable, avec des zones extensibles que vous définissez. Ce type d'image est défini dans un fichier .png avec un format spécial.

Un peu moins à la mode depuis que les standards Material Design ont rendu obsolète les couleur de fond en gradient

Construire une interface responsive

Créer des animations simples

Le frameword Android fournit déjà plusieurs facons d'ajouter des animations dans une application:

- Animated Vector Drawable
- Property Animation framework
- LayoutTransition animations
- Layout transitions with TransitionManager
- CoordinatorLayout

On peut citer plusieurs exemples de widgets ou conteneurs incorporant nativement des animations :

- Drawer
- ViewPager
- CoordinatorLayout

Construire une interface responsive

Créer des animations simples

BottomSheetView

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/bottom_sheet"
    android:layout_width="match_parent"
    android:layout_height="340dp"
    android:background="@android:color/darker_gray"
    android:orientation="vertical"
    app:behavior_hideable="true"
    app:behavior_peekHeight="80dp"
    app:layout_behavior="android.support.design.widget.BottomSheetBehavior">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="80dp"
        android:background="@color/colorAccent"
        android:gravity="center"
        android:text="Ceci est la Peek View"
        android:textColor="@android:color/white" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:text="Ceci est le contenu de la Bottom Sheet"
        android:textColor="@android:color/white" />

</LinearLayout>
```

Construire une interface responsive

Créer des animations simples

BottomSheetView

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Contenu principal"
        android:gravity="center"/>

    <!-- include bottom sheet -->
    <include layout="@layout/bottom_sheet" />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Construire une interface responsive

Créer des animations simples

Shared Element

La « shared element transition » détermine la façon dont les vues présentes dans deux activités/fragments passent entre elles.

Il suffit d'attribuer à ces vues communes un « transitionName », Android gérera la transition

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:transitionName="shared_element"/>
```

Construire une interface responsive

Créer des animations simples

Motion Layout

Un MotionLayout est un ConstraintLayout qui nous permet d'animer les transitions des éléments entre différents états.

Il est entièrement déclaratif et peut s’écrire entièrement en XML

MotionLayout n’est actif que pour ses propres enfants, c’est à dire au sein d’une même activité. Il ne gère donc pas les transitions entre activités.

Construire une interface responsive

Créer des animations simples

Motion Layout

```
<?xml version="1.0" encoding="utf-8"?>
<!-- activity_main.xml -->
<androidx.constraintlayout.motion.widget.MotionLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/motionLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layoutDescription="@xml/scene_01"
    tools:showPaths="true">

    <View
        android:id="@+id/button"
        android:layout_width="64dp"
        android:layout_height="64dp"
        android:background="@color/colorAccent"
        android:text="Button" />

</androidx.constraintlayout.motion.widget.MotionLayout>
```

Construire une interface responsive

Créer des animations simples

Motion Layout

```
<?xml version="1.0" encoding="utf-8"?>
<MotionScene xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:motion="http://schemas.android.com/apk/res-auto">
    <Transition
        motion:constraintSetStart="@+id/start"
        motion:constraintSetEnd="@+id/end"
        motion:duration="1000">
        <OnSwipe
            motion:touchAnchorId="@+id/button"
            motion:touchAnchorSide="right"
            motion:dragDirection="dragRight" />
    </Transition>
    <ConstraintSet android:id="@+id/start">
        <Constraint
            android:id="@+id/button"
            android:layout_width="64dp"
            android:layout_height="64dp"
            android:layout_marginStart="8dp"
            motion:layout_constraintBottom_toBottomOf="parent"
            motion:layout_constraintStart_toStartOf="parent"
            motion:layout_constraintTop_toTopOf="parent" />
    </ConstraintSet>
    <ConstraintSet android:id="@+id/end">
        <Constraint
            android:id="@+id/button"
            android:layout_width="64dp"
            android:layout_height="64dp"
            android:layout_marginEnd="8dp"
            motion:layout_constraintBottom_toBottomOf="parent"
            motion:layout_constraintEnd_toEndOf="parent"
            motion:layout_constraintTop_toTopOf="parent" />
    </ConstraintSet>
</MotionScene>
```

Analyse et optimisation d'UI

- Analyser les performances avec Android Studio
- Garantir la performance des layouts

Analyse et optimisation d'UI

Analyser les performances avec Android Studio

Layout Inspector et Android Profiler

https://developer.android.com/studio/profile

Analyse et optimisation d'UI

Garantir la performance des layouts

Utiliser les compound drawables

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="80dp"
    android:drawableStart="@drawable/vector"
    android:text="Je suis un texte avec image" />
```

Ré-utiliser les layouts avec le tag <include>

```
<include layout="@layout/bottom_sheet" />
```

Créer des ids uniques de préférence

Analyse et optimisation d'UI

Garantir la performance des layouts

Fusionner les layouts parents avec le tag <merge>

```
<!-- children.xml -->
<LinearLayout
    android:id="@+id/layout2"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:id="@+id/textView1" />

    <TextView
        android:id="@+id/textView2"/>
</LinearLayout>
```

```
<!-- parent.xml -->
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <include
        android:id="@+id/children.xml"/>

    <Button
        android:id="@+id/button1" />

</LinearLayout>
```

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <LinearLayout
        android:id="@+id/layout2"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <TextView
            android:id="@+id/textView1" />

        <TextView
            android:id="@+id/textView2"/>
    </LinearLayout>

    <Button
        android:id="@+id/button1" />

</LinearLayout>
```

DOCDOKU digital enterprise applications

Analyse et optimisation d'UI

Garantir la performance des layouts

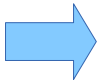
Fusionner les layouts parents avec le tag <merge>

```
<!-- children.xml -->
<merge
xmlns:android="http://schemas.android.com/apk/res/android">

    <TextView
        android:id="@+id/textView1"/>

    <TextView
        android:id="@+id/textView2"/>

</merge>
```



```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/textView1" />

    <TextView
        android:id="@+id/textView2"/>


    <Button
        android:id="@+id/button1" />

</LinearLayout>
```

Analyse et optimisation d'UI

Garantir la performance des layouts

Supprimer les layouts inutiles

Un layout sans enfant ou sans couleur de fond peut souvent être supprimé (car il est invisible)

Supprimer les layouts parents inutiles

Un layout parent n'ayant qu'un seul enfant, sans couleur de fond peut être supprimé pour déplacer l'enfant directement dans le parent et réduire la hiérarchie de vues

Réduire les imbrications

Les mises en page avec trop d'imbrications sont mauvaises pour les performances, la profondeur maximale par défaut étant de 10. Favoriser l'emploi des ConstrainteLayout pour éviter cela.

Eviter les remplissages inutiles

Si un parent est totalement recouvert par ses enfants, il est comme invisible pour l'utilisateur. Il n'est donc pas nécessaire de lui ajouter une image ou couleur de fond.

Architecture et data-binding

- Structurer son application grâce aux « Android Architecture Components »
- Découper l'interface graphique en fragment
- Organiser le flux de données
- View et Data binding

Les fragments

- L’objectif est de modulariser les composants graphiques
- Pour permettre :
  - La réutilisation des éléments UI
  - Diviser la complexité des écrans
- Les fragments sont une solution complète
  - Réutilisation de l’apparence
  - Et également du comportement
  - Gèrent leur propre cycle de vie

Création d'un fragment

1)Étendre la classe Fragment

2)Redéfinir ***onCreateView*** ou appeler ***super(int contentLayoutId)***

3)Définir un *placeholder* dans le layout de l'activité soit avec :

- androidx.fragment.app.FragmentContainerView (recommandé)
- FrameLayout (Standard, inclus dans Android mais bugué)

1)Injecter le fragment dans le *placeholder*

- Spécifier la classe du fragment avec android:name
- Inclure programmatiquement le fragment

Manipulation programmatique des fragments

- Ajout/Suppression par FragmentManager
- Les transactions sont opérées en asynchrone par le Thread UI
- SetReorderingAllowed « compacte » les Tx devant être exécutées
- if (savedInstanceState == null) permet de ne pas refaire la transaction suite à une recréation (les fragments seront automatiquement déjà remis)

```
if (savedInstanceState == null) {
    getSupportFragmentManager().beginTransaction()
            .setReorderingAllowed(true)
            .add(R.id.fragment_container_view,
BlankFragment.class, null)
            .commit();
}
```

Communication entre Activity et Fragment

- Les fragments doivent rester découplés !
  - Pas d'appels directs entre les fragments et les activités

- Les classes ViewModel et LiveData organisent ce bus de communication
  - Les **ViewModel**
    - Gèrent les données des écrans
  - Les **LiveData**
    - Implémentent le pattern observer

ViewModel exemple

- Suivent le même cycle de vie que l'activité ou le fragment spécifié à la création
- Singleton dans le contexte de l'activité ou fragment lié
- Instancié avec un ViewModelProvider
- Optionnellement un ViewModelProvider.Factory peut personnaliser cette instanciation
- Définissent en variables des LiveData

```
//dans l'activité
loginViewModel = new
ViewModelProvider(this).get(LoginViewModel.class);
//dans le fragment
loginViewModel = new
ViewModelProvider(getActivity()).get(LoginViewModel.class);
```

Architecture logicielle – ViewModel et LiveData

- La classe LiveData est un flux de données qui est lui même « lifecycle aware ».
- MutableLiveData fournit une implémentation qui expose :
  - **setValue** (thread courant)
  - **postValue** (main thread)
- La classe **Transformations** dispose de méthodes statiques de composition
- Pour plus de maîtrise **MediatorLiveData**

LiveData exemple

- Offre la gestion *d'observers* notifiés quand les valeurs changent
- Les *Observers* sont liés au cycle de vie d'autres objets (activités et fragments)

```
viewModel.getLiveData().observe(this, liveData -> {
        if (liveData == null) {
            return;
        }
        //...
    });
```

View et Data binding

- Pour éviter le fastidieux « findViewById(R.id.btn) »
- Fonctionne grâce à de la génération de code
- Deux types de binding :
  - **viewBinding** : la classe générée (nom du layout en camel case + « Binding ») référence toutes les view
  - **dataBinding** : plus sophistiqué, le layout xml contient des références vers des objets Java
- Déclenchée en ajoutant dans le build.gradle :

```
buildFeatures {
    dataBinding true
}
```

View binding

- Changement dans la méthode onCreate
- Puis on accède simplement aux views par mBinding.monButton.setOnClickListener...

```
private ActivityMainBinding mBinding;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    mBinding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(mBinding.getRoot());
}
```

Data binding 1/2

- On référence les objets portant les données de la vue directement depuis le layout :

```
<?xml version="1.0" encoding="utf-8"?>
<layout
xmlns:android="http://schemas.android.com/apk/res/android">
<data>
    <variable name="school"
type="com.training.android.tp.School"/>
</data>
}
…
<TextView
    android:id="@+id/textView"
    android:text="@{school.description, default=none}"
```

Data binding 2/2

- Dans le code (méthode onCreate), on indique l'instance qui sera accédée depuis le layout
- Généralement un LiveData, d'où l'appel à setLifecycleOwner

```
@Override
protected void onCreate(Bundle savedInstanceState) {
...

binding.setSchool(this.mySchool);
binding.setLifecycleOwner(this);
}
```

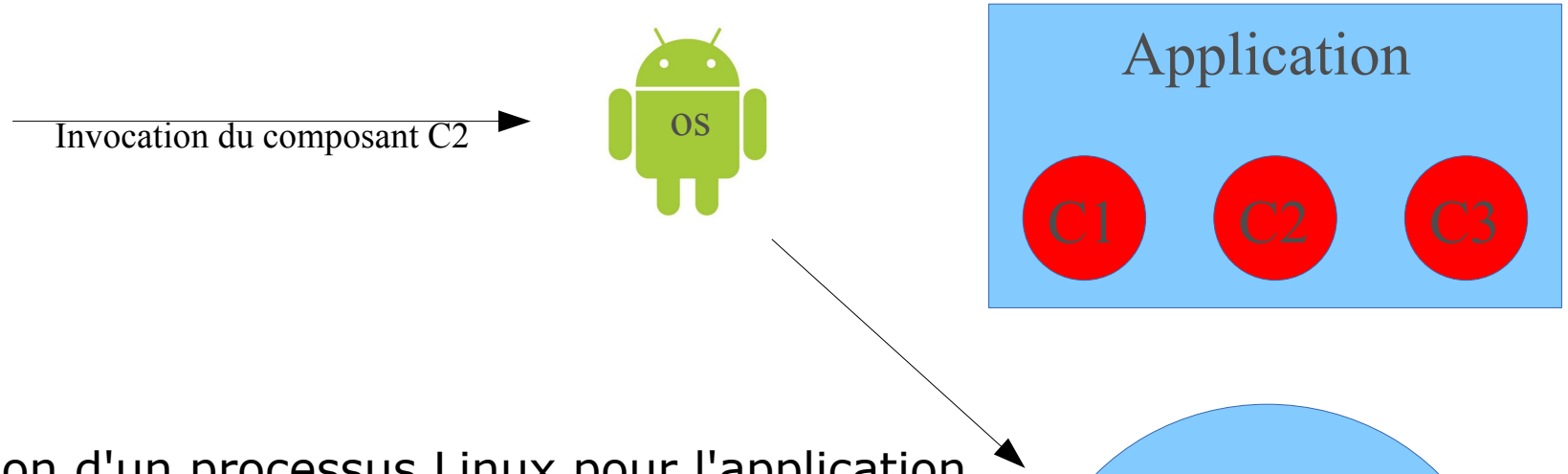
Thread et Programmation réactive

- Rappel : processus et threads
- Traitements asynchrones
- RxJava
- Le JobIntentService

Rappel : processus et threads

Processus et Threads :

1er cas : L'application ne « tourne » pas, aucun composant démarré.



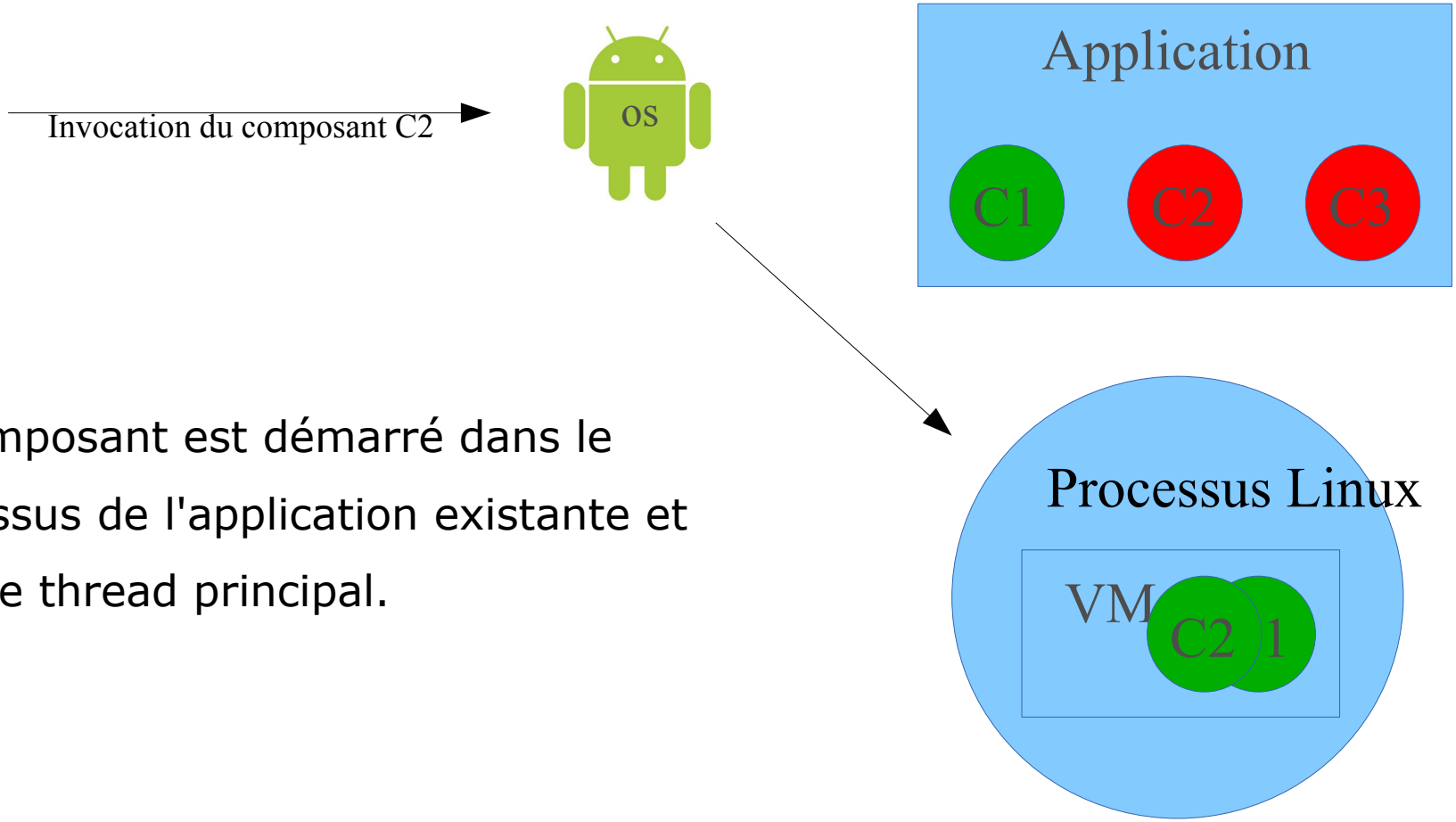
Création d'un processus Linux pour l'application.

Par défaut tous les composants d'une même application tournent dans le même processus et dans le même thread : le « Main Thread ».

Rappel : processus et threads

Processus et Threads :

2ème cas : L'application « tourne », au moins 1 composant démarré.



Le composant est démarré dans le processus de l'application existante et dans le thread principal.

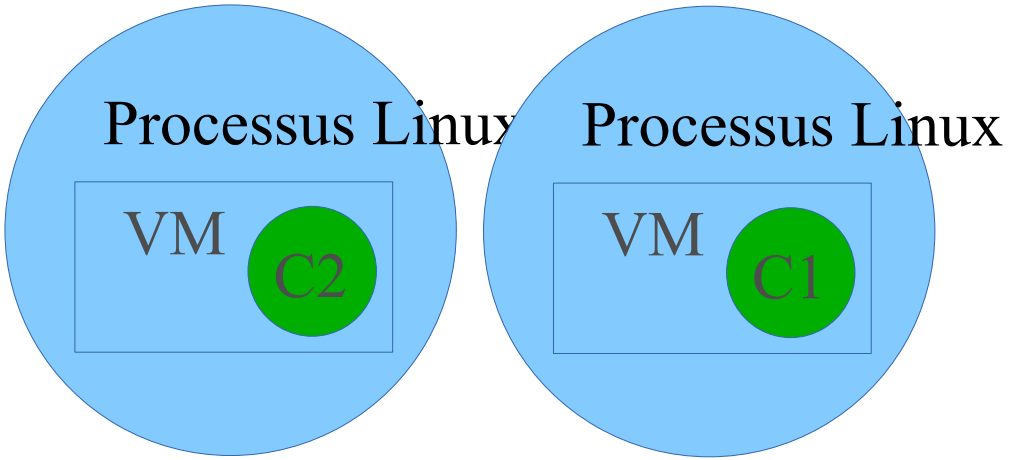
Rappel : processus et threads

Processus et Threads :

2ème cas : L'application « tourne », au moins 1 composant démarré.

Possibilité de lancer chaque composant dans un processus différent grâce au manifeste.

Possibilité de créer de nouveaux threads dans chaque processus.



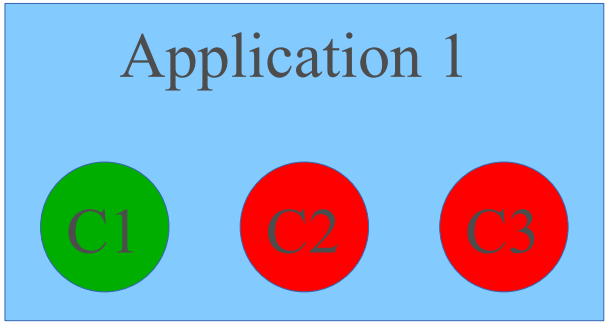
Déconseillé dans les dernières versions !

Rappel : processus et threads

Processus et Threads :

Manifeste : les éléments <activity>, <service>, <receiver> et <provider> ont un attribut `android:process="`*`nom_de_process`*`"`.

- Par défaut, c'est le nom du package de l'application
- Si le nom assigné commence par `":"` le processus est privé à l'application
- Sinon le processus est global, les composants de différentes applications peuvent alors partager un processus.
  - Cela permet de réduire l'utilisation des ressources



Déconseillé dans les dernières versions !

Les threads, la problématique

Problématique des traitements longs:

Deux règles à respecter impérativement :

- Ne pas bloquer le thread qui gère l'interface utilisateur (Main Thread)
- Ne pas accéder à l'interface utilisateur en dehors du Main Thread

Les solutions : worker thread, la classe AsyncTask, les IntentService, les singletons / thread-safe, les ContentProviders, les loaders, les notifications

Beaucoup sont dépréciés aujourd'hui

Worker thread : une solution basique

Solution acceptable mais attention à la prolifération des threads

```
public void onClick(View v) {
      new Thread(new Runnable() {
            public void run() {
                  Bitmap b = loadImageFromNetwork("http://docdoku.com/image.png");
                  mImageView.setImageBitmap(b);
            }
      }).start();
}
```

```
public void onClick(View v) {
      new Thread(new Runnable() {
            public void run() {
                  final Bitmap bitmap = loadImageFromNetwork("http://docdoku.com/image.png");
                  mImageView.post(new Runnable() {
                        public void run() {
                              mImageView.setImageBitmap(bitmap);
                        }
                  });
            }
      }).start();
}
```

Autre solution : AsyncTask

AsyncTask : dépréciée depuis l'API 30

```
public void onClick(View v) {
      new DownloadImageTask().execute("http://docdoku.com/image.png");
}

private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
      /** The system calls this to perform work in a worker thread and
        * delivers it the parameters given to AsyncTask.execute() */
      protected Bitmap doInBackground(String... urls) {
            return loadImageFromNetwork(urls[0]);
      }

      /** The system calls this to perform work in the UI thread and delivers
        * the result from doInBackground() */
      protected void onPostExecute(Bitmap result) {
            mImageView.setImageBitmap(result);
      }
}
```

Handler : accès partout au main thread

- Si on n'a pas accès aux méthodes Activity.runOnUiThread ou View.post
- Handler et Looper offre une solution universelle :

```
mHandler = Handler(Looper.getMainLooper())
mHandler.post {
    // Update UI here
}
```

- Enfin, il est également possible d'utiliser la programmation réactive (lib Rx notamment)
  - Toutefois complexe
  - Pas toujours nécessaire car certaines librairies proposent des mécanismes intégrés (Volley par exemple)

Programmation réactive

- Modèle de programmation pour traiter **flux** de données
  - déclarer a = b + c
  - **a** mis à jour quand sources **changent**

Concepts clés

- **Observable** : flux d'événements
- **Observer** : abonné au flux
- **Operator** : algo qui transforme un flux en un nouveau flux
- **Scheduler** : sert à exécuter les traitements sur différents thread

Observable

- Flux de données auquel on peut **s'abonner**
- Peut émettre 3 types d'événement :
  - **next** : nouvelle valeur
  - **error**: erreur, le flux s'arrête
  - **completed** : terminaison du flux sans erreur

Observer

```
public interface Observer<@NonNull T> {
    void onSubscribe(@NonNull Disposable subscription);
    void onNext(@NonNull T value);
    void onError(@NonNull Throwable error);
    void onComplete();
}
```

Créer un observable

Méthode de base Observable.create

```
class Observable<T> {
  static <T> Observable<T> create(
    ObservableOnSubscribe<T>
source) {
…
  }
 }
```

La source d'événements

```
interface ObservableOnSubscribe<T>
  void subscribe(
    ObservableEmitter<T> emitter);
 }
```

ObservableEmitter

```
interface ObservableEmitter<T> {
  void setDisposable(Disposable subscription);
  void onNext(T value);
  void onError(Throwable error);
  void onComplete();
  }
```

Exemple

Création

```
Observable<String> obs = Observable.create(
  emitter → {
    emitter.onNext("hello");
    emitter.onNext("world");
    emitter.onComplete();
  });
```

Subscription

```
obs.subscribe(new Observer<String>(){
...
    @Override
    public void onNext(@NonNull String s){
    ...
    }
...
  });
```

Plusieurs méthodes subscribe

- **subscribe()**
  - "démarrer" l'observable sans traiter les résultats
- **subscribe(Observer)**
  - l'abonnement générique
- **subscribe(Consumer, Consumer, Action)**
  - permet d'utiliser des lambdas

Plusieurs façons de créer un observable

- **Observable.create**
- **Observable.just**
- **Observable.fromIterable / fromArray**
- **Observable.fromCallable / fromFuture**
- **Observable.interval**
- **Observable.empty / never / error**

Deux modes de création Cold vs Hot

- **Cold observable**
  - source d'événements créée à l'abonnement
  - unicast
  - ex.: requête HTTP

- **Hot observable**
  - source d'événements créée en dehors des abonnements
  - multicast
  - ex.: touch events

Désabonnement

Observer doit se désabonner

- si plus intéressé par l'observable
- si termine avant l'observable

Désabonnement inutile

- quand observable plante ou se termine

Fait grâce à l’objet Disposable

Contrairement aux LiveData, désabonnement n’est pas automatique !

5 types d'observables

- **Observable**
- **Flowable** : un observable avec des capacités de backpressure
- **Single** : 1 seul évènement (1 donnée ou erreur)
- **Maybe** : 1 ou 0 seul évènement (1 donnée, fin ou erreur)
- **Completable** : 1 seul évènement (fin ou erreur)

Les Subjects

- **À la fois un observer et un observable**
- Observable: fournit un subscribe
- Observer: founit onNext, onComplete, ...

```
Subject<String> sub = PublishSubject<String>.create();
subject.subscribe(…);
// Emission d'un événement sur le subjet
subject.onNext("Hello");
```

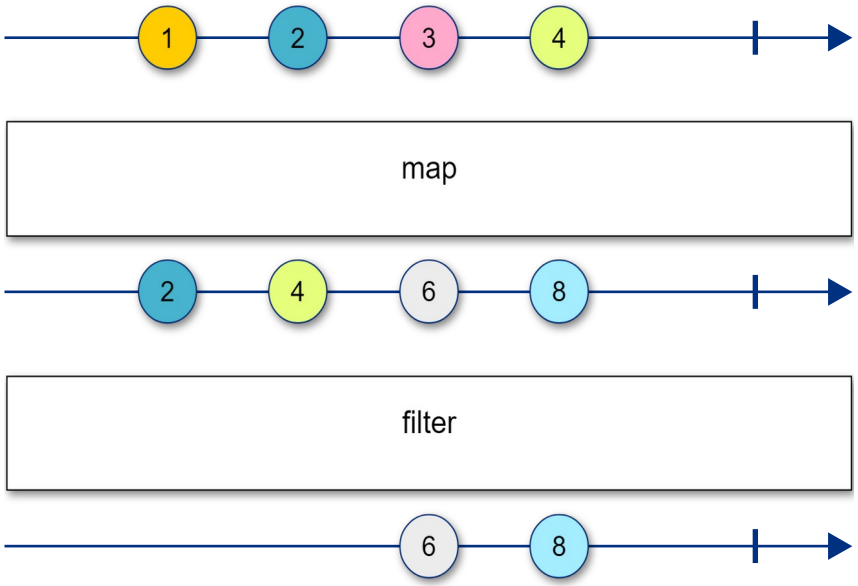
Plusieurs types de subjects

- **PublishSubject** : implémentation de base, peut servir de bus d'évènements
- **BehaviorSubject** : garde la dernière valeur émise, souvent branché à un observable
- **ReplaySubjet** : conserve les dernières n valeurs émises, souvent branché à un observable
- **AsyncSubject** : n'émet que la dernière valeur (à la fin)
- **UnicastSubject** : conserve toutes les valeurs jusqu'à ce qu'un unique observer s'abonne

Observable : opérateurs

- Fonctions de projection **chaînables**

```
Observable.unOperateur(...): Observable
```




```
observable
   .map{ x -> 2 * x }
   .filter{ x -> x > 5 }
   .subscribe()
```

Traitement plus long – décorrélé de l'UI

Besoin :

- Pour les tâches de fonds qui doivent continuer même après que l'utilisateur ait quitté l'application
- **Ne pas oublier** : un processus sans *Activity*, juste avec un worker thread sera brutalement arrêté !

Solution: IntentService et maintenant JobService

- Tourne dans le processus de l'application où il est déclaré
- Indépendant des activités
- Invoqué depuis l'application à laquelle il appartient, ou avant l'API 30 depuis un programme tiers grâce à startService(Intent)

Traitement plus long – décorrélé de l'UI

Explication

- Le service démarre au besoin, gère chaque Intent un par un (file d'attente) en utilisant un unique worker thread
- Il s'arrête automatiquement dès qu'il n'a plus rien à faire
- Opération non interruptible
- Usage non recommandé sous Android 8.0 à cause du système Doze
  - Privilégier JobIntentService (onHandleIntent → onHandleWork)

JobIntentService - implémentation

Implémentation

```
public class CustomService extends IntentService {
    @Override
    protected void onHandleIntent(Intent workIntent) {
        // A implémenter
        ...
    }
}
```

Déclaration dans le manifeste

```
<service
        android:name=".RSSPullService"
        android:exported="false"/>
```

Note : `exported=false` signifie que le service ne sera dispo que sur l'application qui le déclare

L’injection de dépendances

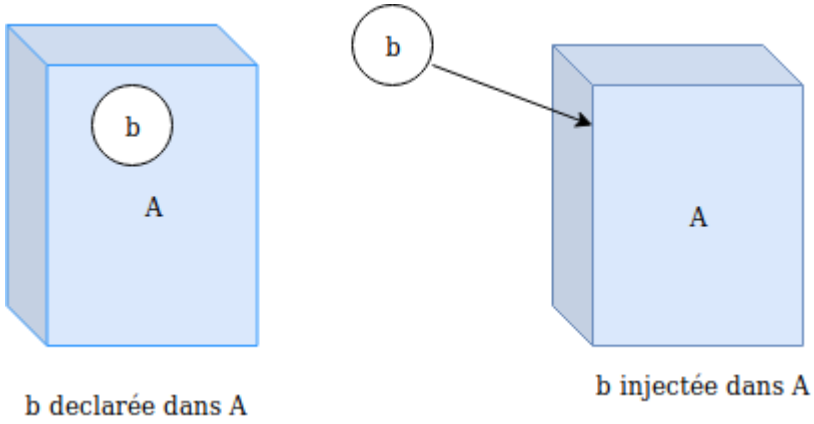
- Injection de dépendances : rappels, enjeux
- Injecter ses dépendances avec la librairie Dagger

L’injection de dépendances

Injection de dépendances - définition

Design pattern pour découpler les dépendances entre objets

Les dépendances sont injectées lors de l’exécution par une entité externe.



L’injection de dépendances

Injection de dépendances - enjeux

```
class MainActivity : AppCompatActivity() {

    var myDependency: MyDependency? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        myDependency = MyDependency()
      }
}
```

L’activité crée elle-même la nouvelle instance de MyDependency

Problèmes :

- Couplage fort entre les deux objets
- Peu de séparation des objets dans des couches applicatives différentes
- Tests unitaires complexes à mettre en place
- Réorganisation et maintenance de code difficile

L’injection de dépendances

Injection de dépendances - enjeux

Comment se libérer de ces couplages ?

↳ Déléguer à un framework l’instanciation des objets à l’exécution

L’injecteur (non plus le développeur) gère :

- la création des objets
- l'appel des méthodes d'initialisation
- les différentes liaisons entre ces objets

L’injection de dépendances

Dagger

- Structure d'annotation simple
- Code généré lisible
- Aucune réflexion (rapide)
- Aucune erreur d'exécution
- Simplification d’utilisation par rapport à la version 1
- Java pur

L’injection de dépendances

Dagger

Exemple d’injection

```
public class MainActivity extends Activity {

    @Inject
    UserData mConnectedUser;
}
```

- A l’usage Dagger est complexe à mettre en œuvre
- Beaucoup de « setup code »
- Pas aussi magique que l’injection coté serveur Java
- Vaut mieux attendre la prochaine version ;-)

Les Tests

Ecrire des tests sert de vérifier :

- L’exactitude du code
- Le comportement fonctionnel
- La facilité d'utilisation
-

Avantages :

- Retour rapide sur les échecs
- Détection précoce des bugs dans le cycle de développement.
- Réfactorisation de code plus sûre
- Vitesse de développement stable, en réduisant la dette technique.

Les Tests

Deux grandes familles de tests :

Unitaires

- Test fonctionnels
- Exécutés dans une JVM (poste de dev ou serveur)
- Fidélité faible
- Maintenance simple et coût de développement faible
- Représentent l'essentiel des tests
- Localisés sous le répertoire **test**

Intégration

- Tests UI
- Exécutés sur un émulateur ou un véritable « device »
- Haute fidélité
- Maintenance plus complexe et coûteux à développer
- A réserver aux scénarios pertinents
- Localisés sous le répertoire **androidTest**

Les Tests, les frameworks

- **Junit** : fournit le cadre général d'exécution
- **Mockito** : création de mocks
- **Hamcrest**, **Truth**, **AssertJ** : librairies d'assertion
- **Espresso** : manipulation programmatique des interfaces graphiques Android

Les Tests d'Intégration

Espresso

Plusieurs composants :

- Espresso
- ViewMatchers
- ViewActions
- ViewAssertions

```
onView(withId(R.id.my_view))
    .perform(click())
    .check(matches(isDisplayed()))
```

ViewMatchers: Id des vues uniques ou ajouter une description

```
onView(allOf(withId(R.id.my_view), withContentDescription("Hello!")))
```

ViewAction: possibilité d'exécuter plusieurs actions à la suite

```
onView(...).perform(typeText("Hello"), click())
```

Les Tests d’intégration

Espresso

```
@Test
fun ensureTextChangesWork() {
    // cherche edittext_docdoku
    onView(withId(R.id.edittext_docdoku))

        // écrit le texte "HELLO" et ferme de clavier
        .perform(typeText("HELLO"), closeSoftKeyboard())

    // presse button_docdoku
    onView(withId(R.id.button_docdoku)).perform(click())

    // vérifie que le contenu de new_text correspond à edittext_docdoku
    onView(withId(R.id.new_text)).check(matches(withText("HELLO")))
}
```

Questions